Available online at www.sciencedirect.com

SCIENCE DIRECT®

Theoretical Computer Science 335 (2005) 3–14

Theoretical Computer Science

www.elsevier.com/locate/tcs

An inexact-suffix-tree-based algorithm for detecting extensible patterns

Abhijit Chattaraj[a,*], Laxmi Parida[b]

[a] *School of Computer Science & Information Technology, RMIT University, Melbourne, Australia*
[b] *Computational Biology Center, IBM T.J. Watson Research Center, Yorktown Heights, NY10598, USA*

Abstract

Given an input sequence of data, a *rigid* pattern is a repeating sequence, possibly interspersed with *dont-care* characters. The data could be a sequence of characters or sets of characters or even real values. In practice, the patterns or motifs of interest are the ones that also allow a variable number of gaps (or dont-care characters): these are patterns with spacers termed *extensible patterns* In a bioinformatics context, similar patterns have also been called flexible patterns or motifs. The extensibility is succinctly defined by a single integer parameter $D \geqslant 1$ which is interpreted as the allowable space to be between 1 and D characters between two successive *solid* characters in a reported motif. We introduce a data structure called the inexact-suffix tree and present an algorithm based on this data structure. This has been tested on primarily biological data such as DNA and protein sequences. However the generality of the system makes it equally applicable in other data mining, clustering, and knowledge extraction applications.

Keywords: Pattern discovery; Motif discovery; Data mining; Flexible patterns; Extensible patterns; Motifs; Suffix trees

1. Introduction

Given a sequence of data, a *rigid* motif is a repeating pattern, possibly interspersed with dont-care characters, that has the same length in every occurrence in the input sequence. Pattern or motif discovery in data is widely used as a means of understanding large volumes

* Corresponding author.
E-mail addresses: abhijit@cs.rmit.edu.au (A. Chattaraj), parida@us.ibm.com (L. Parida).

0304-3975/$ - see front matter © 2005 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2004.12.013

of data such as DNA or protein sequences [11,16,19,14,5]. We refer the reader to [4] for an extensive survey of existing motif discovery algorithms and implementations.

Allowing the motifs to have a variable number of gaps (or dont-care characters), termed patterns with spacers or extensible [1] motifs, further increases the expressibility of the motifs [18,10]. For example given a string $s = abcdaXcdabbcd$, $m = a.cd$ is a rigid pattern that occurs twice in the data at positions 1 and 5 in s. In the above example, the extensible motif where the number of dont-care characters between a and c of the pattern is one or two, would occur three times at positions 1, 5 and 9. At position 9 the dot character represents two gaps instead of one. The definition of the extensible pattern used in this paper is the one described in [13] which in turn is an extension of the *generalized regular pattern* described in [4] in the sense that the discovery algorithm can also handle real-valued input.

The task of discovering patterns must be clearly distinguished from that of matching a given pattern in a database (for instance [1]). In the latter situation we know what we are looking for, while in the former we do not know what is being sought. Typically, the higher the self similarity in the sequence, the greater is the number of patterns or motifs in the data. Sometimes the input is pre-processed using heuristics, to remove the repeating or self-similar portions of the data and at other times a *statistical significance* measure [11,3] is used to discard patterns. However, due to the absence of a good understanding of the domain, there is no consensus over the right model to use. Thus there is a trend towards model-less motif discovery in different fields [16,19,14]: we use the same approach to the pattern discovery problem in this paper.

Finally, we give a small example to convince the reader that allowing a variable number of spacers is useful since a rigid pattern, albeit with dont-care characters, discovery tool is sometimes inadequate in detecting biologically significant motifs. Fibronectin is a plasma protein that binds cell surfaces and various compounds including collagen, fibrin, heparin, DNA, and actin. The major part of the sequence of fibronectin consists of the repetition of three types of domains, which are called type I, II, and III. Type II domain is approximately forty residues long, contains four conserved cysteines involved in disulfide bonds and is part of the collagen-binding region of fibronectin. In fibronectin the type II domain is duplicated. Type II domains have also been found in various other proteins. The fibronectin type II domain pattern has the form shown below:

C..PF.[FYWI].....C-(8,10)WC....[DNSR][FYW]-(3,5)[FYW].[FYWI]C

The extensible part of the pattern is shown as integer intervals. It is clear that a rigid pattern discovery tool will never capture this as a single domain.

We introduce a data structure called the inexact-suffix tree and the algorithm is based on this data structure that also provides a framework for an output sensitive detection algorithm.

Roadmap. We introduce some basic definitions in Sections 2 and 3 relates the patterns with fixed gaps to the patterns with variable gaps. We introduce the inexact-suffix-tree data structure in Section 4, and provide an algorithm to construct this in Section 5. We conclude with some preliminary experimental results in Section 6.

[1] Alberto Apostolico suggested this name as a term more appropriate than ‘flexible’, during my visit to Zentrum fur Interdisziplinare Forschung, Bielefeld: the authors concur.

2. Definitions

The notation and terms used in this section have been used in an earlier paper [13] and have been reproduced here to keep this paper self-contained.

Let s be a sequence of sets of characters from an alphabet Σ, '.' $\notin \Sigma$. The '.' is called a dont-care or a dot character and any other element is called *solid*. Also, σ will refer to a singleton character or a set of characters from Σ.

Definition 1. $(e_1 \preceq e_2)e_1 \preceq e_2$ if and only if e_1 is a dont-care character or $e_1 \subseteq e_2$.

For example, if $e_1 = \{A, C\}$, $e_2 = \{A, C, G\}$ and $e_3 = \{T\}$ are three elements of some sequence, then $e_1 \preceq e_2$ and $e_1 \not\preceq e_3$. Allowing for spacers in a motif is what makes it extensible. Such spacers are indicated by annotating the dot characters.

Definition 2 (*Anotated dot character,* $.^{\alpha}$). An annotated "." character is written as $.^{\alpha}$ where α is a set of positive integers $\{\alpha_1, \alpha_2, \ldots, \alpha_k\}$ or an interval $\alpha = [\alpha_l, \alpha_u]$, representing all integers between α_l and α_u including α_l and α_u.

Definition 3 (*Rigid, extensible string*). Given a string m, if at least one dot element, is annotated, m is called an *extensible* string, otherwise m is called *rigid*.

Definition 4 (*Realization*). Let m be an extensible string. A rigid string m' is a realization of m if each annotated dot element $.^{\alpha}$ is replaced by l dot elements where $l \in \alpha$.

Definition 5 (m *occurs at* l). A rigid string m occurs at position l on s if $m[j] \preceq s[l+j-1]$ holds for $1 \leqslant j \leqslant |m|$. An extensible string m occurs at position l in s if there exists a realization m' of m that occurs at l.

To avoid clutter, the annotation superscript α will be an integer interval. If $m = a.^{[2,4]}b.^{[3,6]}cde$, then $m' = a...b...cde$ is a realization of m and so is $m'' = a...b.....cde$. If m is extensible then m could possibly occur a *multiple number of times* at a location on a string s. For example, if $s = axbcbc$, then $m = a.^{[1,3]}bc$ occurs twice at position 1 as $\mathbf{a}x\mathbf{bc}bc$ and $\mathbf{a}xbc\mathbf{bc}$. In the rest of the discussion we will assume that we are interested in exactly one occurrence (the first possible one) at a location.

Definition 6 (*K-motif* m, *location list* $\mathcal{L}_m$). Given a string s on alphabet Σ and a positive integer k, $K \leqslant |s|$, a string (extensible or rigid) m is a motif with $|m| > 1$ and location list $\mathcal{L}_m = (l_1, l_2, \ldots, l_p)$, if both $m[1]$ and $m[|m|]$ are solid and m occurs at each $l \in \mathcal{L}_m$ with $p \geqslant K$. Also $\mathcal{L}_m$ is complete, i.e., if there exists j such that m occurs at j then $j \in \mathcal{L}_m$.

Definition 7 (*Size of* m, $|m|$). If m is rigid, size of m is the number of solid and dot characters in m and is denoted by $|m|$. If m is extensible and occurs at positions in $\mathcal{L}_m$, then $|m| = \max_i |m'_i|$ where m'_i is a realization of m that occurs at $i \in \mathcal{L}_m$.

To avoid clutter, in the rest of the discussion a K-motif will be referred to simply as a motif. The associated K should be clear from the context. Consider $s = abcdabeed$. Let $m_1 = ab$, $m_2 = ab\text{-}d$. Then $|m_1| = 2$ and $|m_2| = \max\{|ab.d|, |ab..d|\} = 5$.

Definition 8 (*Realization of a motif m of s*). Given a motif m on an input string s with a location list $\mathcal{L}_m$ and m' a realization of the string m, then m' is a realization of the motif m if and only if there exists some $i \in \mathcal{L}_m$ such that m' occurs at i in s.

Notice that because of our notation of annotating a dot character with an integer interval (instead of a set of integers), not *every* realization of the extensible string occurs in the input string. For example for $s = axbcbc$, $p = a.^{[1,3]}b$ is a extensible motif on s. $p' = a..b$ is a realization of the string p but not of the motif p since p' does not occur in s. In the remaining discussion we will use this stricter definition of motif realization (Definition 8) unless otherwise specified.

Definition 9 ($m_1 \preceq m_2$). Given two motifs m_1 and m_2 on s. If both m_1 and m_2 are rigid then $m_1 \preceq m_2$ holds if $|m_1| \leqslant |m_2|$ and $m_1[j] \preceq m_2[j]$, $1 \leqslant j \leqslant |m_1|$. If at least one of m_1, m_2 is not rigid, then $m_1 \preceq m_2$ holds if at each co-occurrence i on s, the realization m'_1 of motif m_1 at i there exists a realization m'_2 of motif m_2 at i such that $m'_1 \preceq m'_2$.

For example, let $m_1 = AB..E$ and $m_2 = ABC.E.G$ then $m_1 \preceq m_2$. Also if $m_1 = AB..E$ and $m_2 = ABC.E\text{-}G$ co-occurring at positions 1 and 15 of $s = ABCXEYGXXABYYEA$ $BCYEYYG$, at position 1, $m'_2 = ABC.E.G$ is the realization of m_2 and $m_1 \preceq m'_2$ at position 1. At position 15, $m'_2 = ABC.E..G$ is the realization of m_2 and $m_1 \preceq m'_2$ at position 15. Hence $m_1 \preceq m_2$.

Definition 10 (*Sub-motifs of motif m*). Given a motif m let $m[j_1]$, $m[j_2]$, $\ldots$ $m[j_l]$ be the l solid elements in the motif m. Then the sub-motifs of m are given as follows: for every j_i, j_t, the sub-motif $m[j_i \ldots j_t]$ is obtained by dropping all the elements before (to the left of) j_i and all elements after (to the right of) j_t in m.

Definition 11 (*Maximal motif*). Let $m_1, m_2, \ldots, m_k$ be the motifs in a string s. A motif m_i is maximal in length if there exists no m_l, $l \neq i$ with $|\mathcal{L}_{m_i}| = |\mathcal{L}_{m_l}|$ and m_i is a sub-motif of m_l. A motif m_i is maximal in composition if no dot character of m_i can be replaced by a solid character that appears in all the locations in $\mathcal{L}_m$. A motif m_i is maximal in extension if no annotated dot character of m_i can be replaced by a fixed length substring (without annotated dot characters) that appears in all the locations in $\mathcal{L}_m$. A maximal motif is maximal in composition, in extension and in length.

The other way of looking at maximality is that m_1 is not maximal with respect to m_2 if all the information about m_2 can be derived from m_1. The following lemma is easy to see.

Lemma 1. *If* $|\mathcal{L}_{m_1}| = |\mathcal{L}_{m_2}|$ *and* $m_1 \preceq m_2$, m_1 *is non-maximal.*

Since $m_1 \preceq m_2$, m_1 also occurs at each occurrence of m_2 and if the number of occurrences is the same for both, clearly m_2 completely defines m_1.

3. Fixed vs. variable spacers

Given a constant d, for rigid motifs it is to be interpreted that the motif can have fixed 1 or 2 or 3 ... or d dots between successive solid characters and for extensible motifs can have

between 1 and d dot characters between successive solid characters. For clarity of notation, instead of using the annotated dot character, $.^{[1,d]}$ in the motif, we use the dash symbol (-) as the *extensible wild card*. Note that '-' $\notin \Sigma$. Thus a motif of the form $a.^{[1,d]}b$ will be simply written as a-b. Also, let $\mathcal{R}$ be the set of all rigid maximal motifs and let $\mathcal{E}$ be the set of all extensible motifs. Note that $\mathcal{E}$ also includes those maximal motifs that are rigid.

Lemma 2. *Given s with k and d. Then if $m_r \in \mathcal{R}$, then there must be $m_f \in \mathcal{E}$ such that m_r is a sub-motif of m_f.*

If $m_r \in \mathcal{E}$, then $m_r = m_f$. Consider the case where $m_r \notin \mathcal{E}$. In that case m_r must have got extended to the left or to the right to produce the extensible motif m_f.

Let $k = 2$, $d = 2$. Consider $s = aycaazxc$, then $\mathcal{E} = \{a\text{–}c\}$ and $\mathcal{R} = \{\}$ with $|\mathcal{E}| > |\mathcal{R}|$. If $s = abycpqdefabzcxdef$ then $\mathcal{R} = \{ab.c, def\}$ and $\mathcal{E} = \{ab.c\text{–}def\}$ with $|\mathcal{R}| > |\mathcal{E}|$. However, we have seen in practice $|\mathcal{E}| \gg |\mathcal{R}|$.

4. Inexact-suffix tree

We introduce a data structure representing a string, called the *inexact-suffix tree* that efficiently stores all the suffixes of maximal extensible patterns with wild cards. The suffix is inexact in the sense that it contains wild cards (including the extensible wild card). We use the following definition from [2].

Definition 12 (*Meet*). Given strings $s_i, 1 \leqslant i \leqslant l$, their *meet* is string s_m if and only if $s_m \preceq s_i$, and there exists no s' such that $s_m \preceq s' \preceq s_i$, for all i.

Clearly, if $l = 1$ in the definition then $s_m = s_i$.

We define the inexact-suffix tree as a labeled tree satisfying certain properties. Given a string s of size n, let \$, $\phi \notin \Sigma$. We terminate s with \$ as s\$ and ϕ is the empty symbol. Let an edge having the ϕ symbol in its label be called an *empty edge*. Let d be the maximum number of dont-care characters between any two consecutive solid characters and let k be the minimum number of times a pattern must occur on s. A k-tree is a tree that stores the k-motifs. Again, to avoid clutter, in the rest of the discussion a k-tree will be referred to simply as a tree. The associated k should be clear from the context. Consider a rooted tree $\mathcal{T}$ [6] with edges labeled by non-empty strings with the following properties:

(1) Each leaf node is labeled by an integer $1 \leqslant l \leqslant n$.
(2) The edge label is a sequence on $\Sigma + \{\text{'.'}, \text{'-'}, \phi\}$.
   (a) All the outgoing edges of the root node are labeled by strings that start with a solid character.
   (b) No two edges out of a node can be labeled with strings that start with the same solid character.
   (c) Each edge label can have at most d consecutive '.' character and at most 1 consecutive '-' character. Also, the last character must be solid.
   (d) Empty symbol is the first character in a label and must be followed by at least one solid character. Further, if $\phi\sigma$ is a label or a prefix of a label on an outgoing edge of node i, then there must exist another outgoing edge with a label starting with $\phi\sigma$.

(3) For an internal node i, let $R(i)$ be the set of integer labels of the leaves reachable from i. Also, let p_i be the label on the path from the root node to node i.
   (a) Let $j \neq i$ be such that $R(j) = R(i)$, then one of the following holds:
      (i) Without loss of generality, j is an ancestor node of i, or
      (ii) $p_j \not\preceq p_i$ and $p_i \not\preceq p_j$.
      If all the patterns are rigid, i.e., no pattern is extensible, then j must be an ancestor node of i.
   (b) There exists a node i such that p_i is (prefix of) the meet of at least k suffixes of s with parameter d.

When $d = 0$, $\mathcal{T}$ is also called a *suffix tree* [8]. The unique path from the root node to the leaf node labeled by integer i, represents $s[i \ldots n]$ that is obtained by traversing from the root node to the leaf node: concatenating the edge labels of this path gives p and $p \preceq s[i \ldots n]$. Notice that the empty symbol ϕ does not change the input string. For instance $a\phi b = ab$.

When the suffix tree is modified so that no node i and its descendant node j are such that $R(i) = R(j)$ and i has a single child j, by merging the nodes and appropriately changing the labels on the edges, the resulting tree is called a *reduced suffix tree* [8].

Lemma 3. *A reduced inexact-suffix tree has the following properties*:
(1) $|R(i)| > 1$ *for all* i.
(2) *Only the incoming edge to a leafnode can be an empty edge.*

Assume that an internal node i has at least two outgoing edges, yet $|R(i)| = 1$. This implies that a motif can occur multiple times at the same location which is a contradiction (see Section 2). Hence $|R(i)| > 1$ for all internal nodes i.

Consider all the k empty outgoing edges of a node i that have a fixed σ following ϕ in the labels with $\{j_1, j_2, \ldots j_k\} \subseteq R(i)$. By Property 2(d), $k > 1$. Clearly, if p_i is the prefix of the meet of $s_{j_1}, s_{j_2}, \ldots s_{j_k}$ then so is $p_i\sigma$. Hence by Property 3(a), there must exist p' such that $p_i \preceq p'$ and p' is the prefix of the meet. Since such a p' already exists in the reduced tree each of these outgoing edges must be incident on the leaf nodes of the tree.

For an illustrative example see Fig. 1.

We state the following result about the inexact-suffix tree described above.

Theorem 1. *Given* s, k *and* d, *there exists a reduced inexact-suffix tree which is unique, ignoring the empty edges.*

Using Properties 3(a–b) each meet corresponds to an internal node and each internal node corresponds to a prefix of the meet of suffixes of s. If such a tree exists, it must be unique (up to reduction). We show the existence of the tree by the following construction. Let $|s| = n$. Consider the set of all (extensible, with parameters d and k) meets of suffixes of s. Construct a compact trie [8] of this set of extensible strings. Next, make the following local changes to the labels in the trie: If the label s_1 of an edge v_1v_2 where v_2 is not the leaf node, ends in a wild card (rigid or extensible), then construct $s_1 = s_2 + s_3$, where $+$ is a concatenation (without overlap) operation such that s_3 contains all the trailing wild cards of s_1. For instance if $s_1 = a..c$- then $s_3 =$ -. Replace s_1 by s_2 and prepend the label of each outgoing edge on v_2 with s_3. It is possible that s_2 is an empty string, in such a case remove

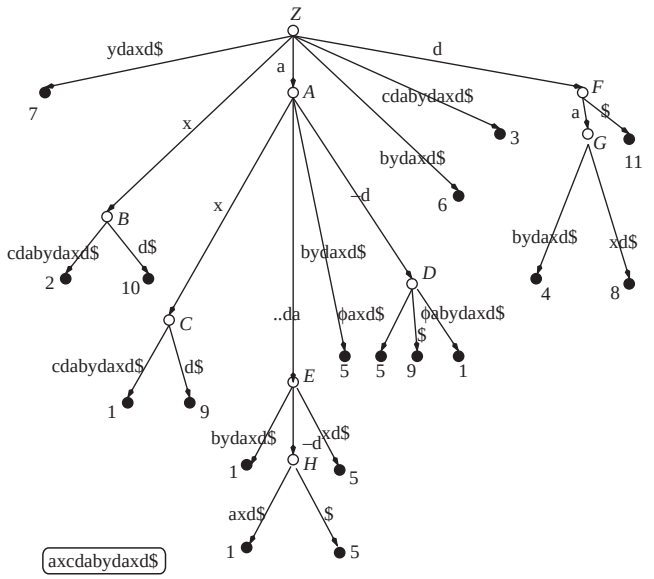

Fig. 1. Reduced inexact-suffix tree for $s = axcdabydaxd$ with $d = 2, k = 2$. A solid circle denotes a leaf node. The root node is labeled Z and the internal nodes are labeled A through H. Notice the use of the empty symbol ϕ to label the outgoing edges of node D that start with the solid character a. Assume there was a node D' with $R(D') = \{1, 5\}$. Then the non-ancestral node E of (hypothetical) node D' has $R(E) = \{1, 5\}$ indicating that a maximal version, of this pattern, a-da, exists. Thus a node such as D' is not created; instead its children are made the children of D with empty edges.

the edge v_1v_2 or merge the vertex v_2 with v_1 that is, all the outgoing edges of v_2 are now the outgoing edges of v_1. Repeat the process until there is no edge label ending in a wild card. Next, we will show that it satisfies the properties of the inexact-suffix tree. Notice that properties 1, 2(a–b) enforce the trie properties in the constructed inexact-suffix tree. The reduction of the tree satisfies condition 2(c). Property 3(b) holds since we started with all possible meets of suffixes as s. Hence the inexact-suffix tree exists and is unique, ignoring the empty edges.

The inexact-suffix tree not only suggests a way of detecting all the extensible patterns efficiently but also gives a data structure for storing the extensible patterns for efficient retrieval or matching.

5. Constructing the inexact-suffix tree

The next question to address is whether the inexact-suffix tree of the last section can be constructed efficiently. The tree is constructed as a reduced suffix-tree as described in [8] with two modifications described below. These ensure that the conditions of the inexact-suffix tree discussed in the last section can be met efficiently.

(1) In the depth first traversal (DFS) a specific order of the children of a node is used.

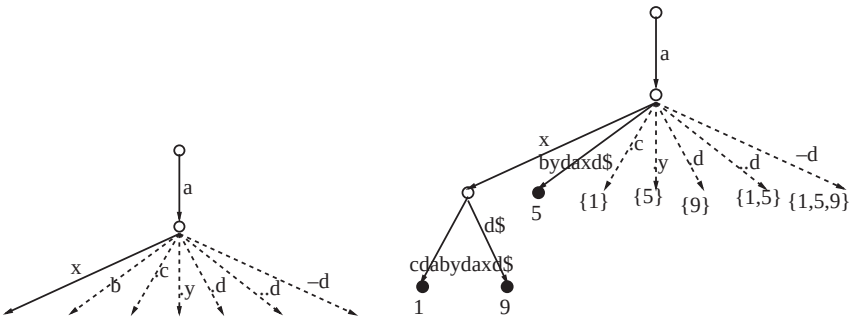

Fig. 2. Inexact-suffix-tree construction for $s = axcdabydaxd$ with $d = 2, k = 2$. For clarity of exposition, we track the subtree starting with the node with label "a". Step 1 (left): the children of the node are shown in the ordering described in Section 5. Step 2 (right): Depth first traversal is carried on with leftmost node(s) and computation of the $R(i)$ for the other nodes.

(2) At each node *i*, a comparison is made with the earlier nodes and a backtracking of at most one level is carried out.

$\preceq$-Ordering of the nodes. Let the children of *i* be $j_1, j_2, j_3, \ldots, j_c$. We claim that $c \leqslant (d+2)\Sigma$ when $d > 1$ and $c \leqslant (d+1)\Sigma$, otherwise. Consider an ordering of the children (called the $\preceq$-ordering) such that the prefixes of the labels on the edges are ordered as follows:

"σ_0", "$.\sigma_1$", "$..\sigma_2$", "$...\sigma_3$", ..., "$.(d$ times$)\sigma_d$"

There can be no more than *d distinct* solid trailing characters in the prefixes above. Thus there can be no more than *d* prefixes of the form "-σ". Augment the ordered list by inserting a prefix of the form "-σ'" after the last occurrence of the prefix in the above ordering with a trailing σ'. For convenience, let $R(label)$ denote $R(i)$ where i is the node reached by following the edge label *label*. Notice that $R(\text{-}\sigma') = \cup R(*\sigma')$, where $*$ denotes 1 to *d* wild cards.

The subtrees rooted at each of the child nodes is constructed in the order defined by the augmented prefix list above. For example, the following is a valid ordering of prefix labels.

"a", ".b", ".c", "..b", "-b", "...d", "....c", "-c"

This ordering ensures that maximal substrings are constructed first during the depth first traversal. See Step 1 in Fig. 2 for the ordering of the children in the example under discussion.

The following lemma about the ordering of the children of a node is straightforward to see.

Lemma 4. *Consider an internal node i in the inexact-suffix tree with children $j_1, j_2, \ldots, j_c$ in the $\preceq$-ordering and let the labels be $p_1, p_2, \ldots p_c$ respectively, then if $p_{j_1} \preceq p_{j_2}$ for some j_1 and j_2 then $j_1 > j_2$ i.e., node j_2 precedes j_1 in the $\preceq$-ordering.*

1-level backtracking. As each node i is traversed, first $R(i)$ is computed and checked against the precomputed sets Let $R(j) = R(i)$ for some existing node j. Further if $p_i \preceq p_j$

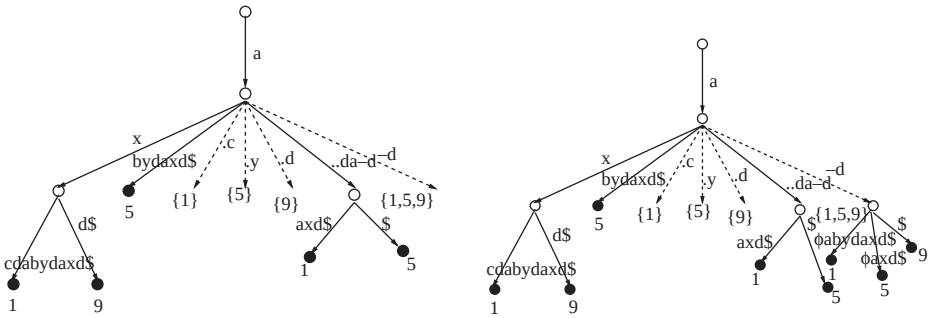

Fig. 3. See example from Fig. 2. Step 3 (left): The leftmost non-singleton node i with $R(i) = \{1, 5\}$ is explored further by a DFS traversal and this subtree is reduced. Step 4 (right): The rightmost node i with $R(i) = \{1, 5, 9\}$ is explored further. Since the set $\{1, 5\}$ already appears (with $a\text{-}da \preceq a..da\text{-}d$), the empty symbol ϕ is used at the rightmost sibling instead of creating an internal node j with $R(j) = \{1, 5\}$).

then

(1) (Non-ancestral) If j is a non-ancestral node of i then this node i is merged with its immediate ancestor k and all the outgoing edges of node i become the empty outgoing edges of k (see Fig. 3).
(2) (Ancestral) If j is an ancestral node, then it must be the immediate ancestor and the node is merged with j if required and the labels appropriately modified.

See Figs. 2 and 3 for a complete example.

Lemma 5. *The backtracking is for only one level at each step.*

This is critical for the efficiency of the algorithm, since large segments of the tree are not destroyed at this step. This follows from Lemma 4.

Lemma 6. *Given a node i the number of times a backtrack*
(1) *due to non-ancestral node can occur is* $O(1)$;
(2) *due to ancestral node can occur is* $O(|p_i|)$.

Non-ancestral node: Let c be the number of children of a node. Then $c = O(1)$, treating $|\Sigma|$ and d as constants. Further, the outgoing edges that are assigned empty symbols are not explored any further. Ancestral node: Clearly, this can occur no more than the number of distinct suffixes of p_i.

5.1. Complexity of the algorithm

If we restrict the maximal patterns to be rigid (not extensible), it has been demonstrated earlier that the number of such patterns can be exponential, in the worst case, in the size of the input [13,14]. So it trivially follows that the number of possible maximal extensible motifs in the worst case can be $O(2^n)$ where n is the size of the input string.

Freq	Seq	Motif	Occurrences
2	2	DVHG.....GMLCAGFLEGGTD	(5:85-106); (8:0-21);
2	2	H-L...ML-F-A-D.....V.E	(1:72-105); (8:2-34);
2	2	F-E.S.....V..I.....P..H-F...ML-F-D	(1:45-97); (5:59-106);
2	2	F-E.S.....V..I.....P..H....T-ML-F-D	(1:45-97); (5:59-106);
2	2	H-S-ML-F-A-D.....V.E	(1:72-105); (8:2-34);
2	2	ML-F-D-S-E	(1:82-104); (8:10-34);
3	3	ML-F-D	(1:82-97); (5:95-106); (8:10-21);
2	2	C-L-R-T....T.S-MK-V-D.....V	(1:3-58); (2:60-95);
2	2	RC-H-T..L-E-ME-E-L.....L-Y.....T-D-T	(0:42-105); (2:3-72);
2	2	E.....F...E.S..LQEA-I.P.RC-H-T-L-L-G-T	(0:10-62); (5:51-105);
2	2	L-Q.....RC-P...G-LC...L-D	(4:66-101); (5:68-106);
2	2	G-S.LQ.....L-RC-P...G-LC...L-D	(4:68-101); (5:61-106);
2	2	K-K...S-N-L.....L-E-RC.....L-L	(0:2-53); (4:46-95);
2	2	E.....F...E.S..LQE-P-RC-H-T-L-L-G-T	(0:10-62); (5:51-105);
2	2	RC...D.....Q...C-E	(4:0-19); (5:80-102);

Fig. 4. The input data is a collection of fibronectin sequences with $d = 7$ and $k = 2$. A small sample output is shown in the table. The first column gives the number of occurrences of the motif shown in the third column; the second column gives the number of distinct sequences in which the motif appears and the last column gives the occurrence in the format $(s : i_1 - i_2)$, where s is the sequence number and the motif starts at i_1 ending at i_2.

Let M be the total number of maximal extensible motifs. Let $P = \sum_{i=1}^{M} |m_i|$ and $L = \sum_{i=1}^{M} |\mathcal{L}_{m_i}|$. Thus the size of the output is $P + L$. We are accounting for the length of each maximal pattern as well as the number of occurrences of each pattern.

Let T be the number of leafnodes in the inexact-suffix tree, then clearly $T \leqslant \sum_m |\mathcal{L}_m|$. Also the number of internal nodes is no more than P (as the suffix of a maximal extensible motif also occupies an internal node). By Lemma 6, the total number of backtracks is bounded by $O(P)$. Also it needs to be checked against the others that can be done in $O(M \log M)$ time, where M is the total number of maximal motifs. The total number of nodes (including leaf nodes) explored by the algorithm is bounded by $P + L$. Thus the algorithm runs in $O(M \log M P^2 + L)$ time.

6. Preliminary experimental results

We present some sample output of test runs on fibronectin data. Fig. 4 shows a small subset of the maximal extensible patterns and their location lists. Many amino acids in protein sequences, are easily interchanged by evolution without loss of function [7]. The use of similarity matrices in the context of DNA sequences such as PAM [17] and BLOSUM [9] is common. The score between two amino acids is higher the more similar they are. It can be either positive or negative. Thus certain amino acids can be grouped together called the homology groups. Fig. 5 shows the extensible motifs using homology groups and their location lists are suppressed in the figure to avoid clutter. Often the different extensible wild card in the pattern correspond to different number of extensible gaps and this information is useful for the biologist. Fig. 6 displays annotated extensible wild card in the patterns on the data. The location list is again suppressed to avoid clutter in the figure.

7. Conclusion

Allowing motifs to have a variable number of gaps (or dont-care characters), i.e., patterns with spacers or extensible motifs, considerably increases the expressibility of the motifs. It

Freq	Seq	Motif
2	2	[FYWI]-[DNSR].....[DNSR]G-[DNSR]PF-S.....A-E-C-H.....Y.....E-[FYWI]-A.....A-T-F-T
2	2	[DNSR]....M-F-D-D.....V-E.....L-[FYWI].....N.PG-YT.V-L...[DNSR]...V
2	2	[DNSR]....M-F-D-D.....V.....[DNSR][DNSR].....I...S-PG-YT.V-L...[DNSR]...V
2	2	[DNSR]....M-F-D-D-V-[DNSR]...L-S-PG-YT.V-L...[DNSR]...V
2	2	V-S-PG-L.....E..T.....S.....[DNSR]....[DNSR]-G..[DNSR]-D-Y.D-Y
2	2	[DNSR]....M-F-D-D-V-[DNSR]...L..I-S-PG-YT.V-L...[DNSR]...V
2	2	S-PG-C-T.....D.....V-[DNSR]-R[DNSR]-T.....[DNSR]..[DNSR]
2	2	S-PG-[FYWI]-L-E-T-S
2	2	S-PG-[FYWI]-E..T.....S.....[DNSR]....[DNSR]-G..[DNSR]-D-Y.D-Y
2	2	S-PG-[FYWI]-D.....G...C.....R[DNSR]-T.....[DNSR]..[DNSR]
2	2	S-PG-[FYWI]-D-G-S.....[DNSR]....[DNSR]-G..[DNSR]-D-Y.D-Y
2	2	S-PG-[FYWI]-D-D-G-C.....R[DNSR]-T.....[DNSR]..[DNSR]
2	2	S-PG-[FYWI]-D-A....[FYWI]-T-[DNSR]
3	2	S-PG-[FYWI]-D
2	2	A-PI.....L-L.....[DNSR]-H
2	2	A-S..TY....A-L....A-C....[DNSR].[FYWI].....[DNSR].....E-C..A-Q-S-VPL...Q-A

Fig. 5. The input data is a collection of fibronectin sequences with $d = 7$ and $k = 2$. A small sample output is shown in the table. The first column gives the number of occurrences of the motif shown in the third column; the second column gives the number of distinct sequences in which the motif appears. This version uses homologous grouping of the amino acid bases shown in square brackets. The occurrence lists have been removed to avoid clutter.

[DNSR]....M-(4,6)F-(2,5)D-(4,6)D.....V-(1,4)E.....L-(2,7)[FYWI].....N.PG-(1,2)YT.V-(1,3)L...[DNSR]...V
[DNSR]....M-(4,6)F-(2,5)D-(4,6)D.....V.....[DNSR][DNSR].....I...S-(5,7)PG-(1,2)YT.V-(1,3)L...[DNSR]...V
[DNSR]....M-(4,6)F-(2,5)D-(4,6)D-(5)V-(7)[DNSR]...L-(6,7)S-(5,7)PG-(1,2)YT.V-(1,3)L...[DNSR]...V
V-(2,7)S-(3,5)PG-(1,4)L.....E..T.....S.....[DNSR]....[DNSR]-(1,5)G..[DNSR]-(1,6)D-(1,6)Y.D-(1,2)Y
[DNSR]....M-(4,6)F-(2,5)D-(4,6)D-(5)V-(7)[DNSR]...L..I-(3,4)S-(5,7)PG-(1,2)YT.V-(1,3)L...[DNSR]...V
S-(3,5)PG-(1,2)C-(1,7)T.....D.....V-(2,4)[DNSR]-(4,7)R[DNSR]-(1,4)T.....[DNSR]..[DNSR]
S-(5,7)PG-(1,2)[FYWI]-(1,6)L-(4,5)E-(1,2)T-(1,6)S
S-(3,5)PG-(2,5)[FYWI]-(1,7)E..T.....S.....[DNSR]....[DNSR]-(1,5)G..[DNSR]-(1,6)D-(1,6)Y.D-(1,2)Y
S-(3,5)PG-(2,5)[FYWI]-(5,6)D.....G...C.....R[DNSR]-(1,4)T.....[DNSR]..[DNSR]
S-(3,5)PG-(2,5)[FYWI]-(5,6)D-(3,7)G-(1,2)S.....[DNSR]....[DNSR]-(1,5)G..[DNSR]-(1,6)D-(1,6)Y.D-(1,2)Y
S-(3,5)PG-(2,5)[FYWI]-(5,6)D-(1,4)D-(1,5)G-(3,4)C.....R[DNSR]-(1,4)T.....[DNSR]..[DNSR]
S-(3,5,7)PG-(1,2,5)[FYWI]-(1,5,6)D
A-(2,6)PI.....L-(1,2)L.....[DNSR]-(3,4)H

Fig. 6. The input data is a collection of fibronectin sequences with $d = 7$ and $k = 2$. A small sample output is shown in the table. Here the gaps are annotated: -(i_1, i_2) indicates that the number of gaps are between i_1 and i_2 in the occurrences in the input.

is likely that some information missed by rigid motifs is captured by the extensible motifs. *Varun* is an implementation of an extensible motif discovery algorithm that guarantees the detection of every extensible pattern. One of the directions being currently investigated is to use Varun to detect extensible patterns in an unsupervised manner on protein sequence databases, and then use suitable pruning techniques to compare the detected patterns with known motifs.

Acknowledgements

The authors would like to thank the anonymous referees for a meticulous feedback that substantially improved the paper.

References

[1] A. Apostolico, M.J. Atallah, Compact recognizers of episode sequences, Inform. Comput. 174 (2002) 180–192.
[2] A. Apostolico, L. Parida, Incremental paradigms for motif discovery, J. Comput. Biol. 11 (4) (2004) 15–25.
[3] T.L. Bailey, M. Gribskov, Methods and statistics for combining motif match scores, J. Comput. Biol. 5 (1998) 211–221.
[4] A. Brazma, I. Jonassen, I. Eidhammer, D. Gilbert, Approaches to the automatic discovery of patterns in biosequences, J. Comput. Biol. 5 (2) (1998) 279–305.
[5] A. Califano, SPLASH: structural pattern localization algorithm by sequential histogramming, Bioinformatics 16 (4) (2000) 341–357.
[6] T.H. Cormen, C.E. Leiserson, R.L. Rivest, Introduction to Algorithms, The MIT Press, Cambridge, MA, 1990.
[7] M.O. Dayhoff, R.M. Schwartz, B.C. Orcutt, A model of evolutionary change in proteins, Atlas of Protein Sequence and Structure, 1978, pp. 345–352.
[8] D. Gusfield, Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology, Cambridge University Press, New York, 1997.
[9] S. Henikoff, J.G. Henikoff, Amino acid substitution matrices from protein blocks, Proc. Natl. Acad. Sci. 89 (1992) 10 915 –10.
[10] I. Jonassen, Efficient discovery of conserved patterns using a pattern graph, CABIOS 13 (1997) 509–522.
[11] I.J.F. Jonassen, J.F. Collins, D.G. Higgins, Finding flexible patterns in unaligned protein sequences, Protein Science, 1995, pp. 1587–1595.
[13] L. Parida, Some results on flexible-pattern matching, in: Proc. 11th Symp. on Comp. Pattern Matching, Lecture Notes in Computer Science, Vol. 1848, Springer, Berlin, 2000, pp. 33–45.
[14] I. Rigoutsos, A. Floratos, Motif discovery in biological sequences without alignment or enumeration, in: Proc. Ann. Conf. on Computational Molecular Biology (RECOMB98), ACM Press, New York, 1998, pp. 221–227.
[16] M.F. Sagot, A. Viari, A double combinatorial approach to discovering patterns in biological sequences, in: Proc. 7th Symp. on combinatorial pattern matching, 1996, pp. 186–208.
[17] R.M. Schwartz, M.O. Dayhoff, Matrices for detecting distance relationships, Atlas of Protein Sequence and Structure, 1978, pp. 353–358.
[18] M. Suyama, T. Nishioka, O. Juníchi, Searching for common sequence patterns among distantly related proteins, Protein Engineering, 1995, pp. 366–385.
[19] J. Wang, G. Chirn, T.G. Marr, B.A. Shapiro, D. Shasha, K. Jhang, Combinatorial pattern discovery for scientific data: some preliminary results, in: Proc. ACM SIGMOD Conf. on Management of Data, 1996, pp. 115–124.